# Improving the Efficiency of Pointer Analysis

***Dissertation submitted to***
***Visvesvaraya National Institute of Technology, Nagpur***
***in partial fulfillment of the requirements for the award of***
***the degree***

## Master of Technology

## in

## Computer Science

*by*

**Ganesan S (MT23MCS007)**

Under the guidance of

**Dr. Swati Ashutosh Ganeshpuri Jaiswal**

**Department of Computer Science and Engineering**
**Visvesvaraya National Institute of Technology**
**Nagpur 440 010 (India)**

**JUNE 2025**

# Improving the Efficiency of Pointer Analysis

***Dissertation submitted to***
***Visvesvaraya National Institute of Technology, Nagpur***
***in partial fulfillment of the requirements for the award of***
***the degree***

## Master of Technology

## in

## Computer Science

*by*

**Ganesan S (MT23MCS007)**

Under the guidance of

**Dr. Swati Ashutosh Ganeshpuri Jaiswal**

**Department of Computer Science and Engineering**
**Visvesvaraya National Institute of Technology**
**Nagpur 440 010 (India)**

**JUNE 2025**

©Visvesvaraya National Institute of Technology (VNIT) 2025

**Department of Computer Science and Engineering**
**Visvesvaraya National Institute of Technology, Nagpur**

# Declaration

I, **Ganesan S**, hereby declare that this dissertation titled **Improving the Efficiency of Pointer Analysis** is carried out by me in the **Department of Computer Science and Engineering** of Visvesvaraya National Institute of Technology, Nagpur. The work is original and has not been submitted earlier whole or in part for the award of any degree/diploma at this or any other Institution / University.

**Ganesan S**
Date:

# Certification

This is to certify that the dissertion titled **Improving the Efficiency of Pointer Analysis** submitted by **Ganesan S** is partial fulfillment of the requirements for the award of the degree the **Master of Technology** in VNIT Nagpur. The work is comprehensive, complete and fit for final evaluation.

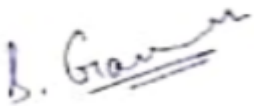
**Dr. Swati Jaiswal**
Assistant Professor, CSE, VNIT Nagpur

**Dr. M. P. Kurhekar**
Head, Department of Computer Science and Engineering
VNIT, Nagpur
Date:

# Department of Computer Science & Engineering
# Visvesvaraya National Institute of Technology, Nagpur

## Non-Plagiarism Certificate

Certified that M. Tech Dissertation titled **"Improving the Efficiency of Pointer Analysis"** submitted by **Ganesan S**, Enrollment No. **MT23MCS007**, ID No. **30132** has been checked for plagiarism using TURNITIN software and the overlap is found within prescribed limits. A summary of the report is given below.



**Ganesan S**
Enrollment No: MT23MCS007

**Dr. Swati Jaiswal**
Ass Professor
Department of Computer Science & Engineering
VNIT, Nagpur

# ACKNOWLEDGEMENTS

I want to express my heartfelt gratitude and sincere appreciation to everyone who has overtly or covertly supported and guided me throughout my M.Tech journey.

I find myself fortunate to be guided by **Dr. Swati Jaiswal** Madam. Her invaluable expertise and guidance along with his constant encouragement and support in all challenging situations, have been instrumental in shaping this thesis. I will always be grateful to be chosen to take up this research under her guidance.

I would like to heartfully thank the faculty members of the **Department of Computer Science and Engineering** at Visvesvaraya National Institute of Technology Nagpur as they have showered me with knowledge and confidence to deal with any oncoming challenges in my life. I am indebted to them for their teachings and guidance which have broadened my perspective and enhanced my understanding of the subject. My sincere appreciation goes to them for their assistance, collaboration, and for providing the necessary resources for conducting experiments and carrying out this research work. Their contributions have been significant in the successful completion of this thesis.

I am indebted to my family and friends especially my mother S.Selva Kumari, my father M.Sivalingam and my friends Divyabharathi Rajasekaran and Boya Aarya Devarla for they have always been there for me, through the highs and the lows of this journey. Their love and motivation have been the driving force behind my accomplishments. Thank you all for being a part of this journey.

**Ganesan S**

# ABSTRACT

Pointer analysis is a fundamental program analysis, which supports diverse use cases such as compiler transformation, error detection, and security scrutiny. However, the trade-off of precision and scalability is still a challenging task, especially for large codebases. Liveness-based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) is a potential solution, which restricts pointer information to live pointers only for better performance. However, L-FCPA struggles to address pointer indirections that frequently need multiple rounds of analysis and do not scale easily. This work introduces an improved L-FCPA, which reduces the number of analysis iterations needed to model pointer indirections. In particular, this thesis offers the following contributions, using flow-insensitive quick points-to data computed by SVF to i) handle indirections on RHS and function pointer calls by raising liveness of pointees provided by SVF. ii) Decision on strong or weak update taken by looking at the pointees provided by SVF. A thorough evaluation of our approach on the SPEC CPU 2006 benchmarks shows reduction in analysis time and rounds with no precision loss. The combination of the flow-insensitive points-to information of SVF makes it possible to build up the needed liveness information in one round. Experimental results on well-known benchmark programs show that we can significantly reduce analysis time and number of rounds with the enhanced L-FCPA compared to the original one and still keeps the same precision. This work proposes an efficient and scalable pointer analysis approach. The potential impact of proposed changes to L-FCPA is considerable, ranging from compiler optimizations to security analysis, through more efficient and precise analysis of software systems.

**Keywords**: Pointer Analysis, Liveness Analysis, Flow-Sensitive Analysis, Context-Sensitive Analysis, Static Analysis, Program Analysis, SVF, Weak Updates, Flow-Insensitive Analysis, L-FCPA, Efficiency, Optimization, Analysis Rounds, Scalability

# Contents


CERTIFICATE . . . . . iii
NON-PLAGIARISM CERTIFICATE . . . . . iv
ACKNOWLEDGEMENTS . . . . . v
ABSTRACT . . . . . vi

**1 Introduction** **1**
1.1 Pointer Analysis . . . . . 1
1.2 L-FCPA . . . . . 2
1.3 Motivation . . . . . 2
1.4 Problem Statement . . . . . 4
1.5 Thesis Outline . . . . . 4

**2 Literature Review** **6**
2.1 Pointer Analysis Techniques . . . . . 6
2.2 Liveness Analysis in Pointer Analysis . . . . . 7
2.2.1 Challenges in Flow- and Context-Sensitive Analysis . . . . . 7
2.2.2 Andersen's Flow-Insensitive Pointer Analysis . . . . . 8
2.3 SVF . . . . . 8
2.4 Summary of Related Work . . . . . 9

**3 L-FCPA - Liveness Based Flow and Context Sensitive Pointer Analysis** **10**
3.1 L-FCPA . . . . . 10
3.2 L-FCPA when there are indirections on the LHS . . . . . 10
3.3 L-FCPA Equations . . . . . 11
3.3.1 Liveness Analysis Equations . . . . . 11
3.3.2 Points-To Analysis Equations . . . . . 12
3.3.3 Extractor Functions . . . . . 13
3.3.4 Data Flow Framework . . . . . 13

**4 SVF - Static Value Flow Analysis Framework** **15**
4.1 Introduction to SVF . . . . . 15
4.2 SVF Input-Output Example . . . . . 16

4.2.1 Input Program . . . 17
4.2.2 SVF Analysis and Output . . . 17
4.2.3 Points-to Information Analysis . . . 18
4.3 PTA Exporter . . . 19
4.3.1 PTA Exporter API . . . 20
4.4 Changes Done to the Header File Format . . . 20

**5 Proposed Solution** **22**
5.1 Motivation . . . 22
5.2 Weak Update for LHS Indirections . . . 23
5.3 SVF Integration for RHS Indirections . . . 24
5.4 Handling Function Pointers with SVF Data . . . 24
5.5 Modified L-FCPA Equations . . . 25
5.5.1 Original L-FCPA Equations for Indirections . . . 25
5.5.2 Modified Equations for LHS Indirections . . . 26
5.5.3 Modified Equations for RHS Indirections . . . 26
5.6 Single-Round Analysis Framework . . . 26
5.7 Summary of Proposed Modifications . . . 27

**6 Implementation Details** **28**
6.1 Architecture . . . 28
6.1.1 Step 1: Generating LLVM IR from C Source Code . . . 29
6.1.2 Step 2: Extracting Points-To Information with SVF . . . 29
6.1.3 Step 3: Feeding Points-To Information into L-FCPA . . . 29
6.1.4 Step 4: Producing Analysis Output . . . 30
6.2 Initialization of Variable Mapping and Points-To Data . . . 30
6.2.1 Extracting and Storing Variable Information . . . 30
6.2.2 Populating the Points-To Set . . . 31
6.2.3 Performing L-FCPA Analysis . . . 31
6.3 Enhanced Liveness Collection for Incoming Operands . . . 31
6.4 Improved Points-to Propagation for Pointer Indirections . . . 32
6.5 Handling Function Pointers . . . 34
6.6 Modifications done to PTA Exporter . . . 34
6.6.1 Data Structure Modification for Node Mapping . . . 34

**7 Empirical Evaluations** **37**
7.1 Experimental Setup . . . 37
7.2 Results and Analysis . . . 38
7.2.1 Analysis . . . 38
7.3 Summary . . . 39

**8 Conclusion** **40**
8.1 Summary of Findings....40

**A LLVM: Architecture and Installation** **42**
A.0.1 LLVM Architecture....42
A.0.2 Steps to Install LLVM....43

**B SLIM IR** **44**
B.0.1 Steps to Install SLIM Library....44
B.0.2 Using SLIM IR....45
B.0.3 Building SLIM....45
B.0.4 Generating SLIM IR from a C File....45
B.0.5 Output Interpretation....46

**C Steps to Install SVF** **47**
C.0.1 Installation Steps....47
C.1 Steps to Install and Use PTA Exporter....48
C.1.1 Installation and Usage Steps....48

**Bibliography** **52**

# List of Figures


1.1 Pointer Analysis ..... 2
1.2 Liveness based pointer analysis ..... 3
1.3 LFCPA with indirection on RHS ..... 3

3.1 LFCPA with indirection on LHS ..... 11
3.2 Data flow framework of L-FCPA Liveness and points-to analysis interaction ..... 13

4.1 SVF analysis workflow From LLVM IR to points-to sets..... 16
4.2 Simplified SVF points-to graph for the input program Showing pointer relationships ..... 18

5.1 Modified L-FCPA by doing weak update..... 23
5.2 Example illustrating modified L-FCPA for RHS indirections Statement x = *y 24

6.1 Flow chart depicting the flow of program ..... 29

# List of Tables

4.1 Summary of pointer relationships from SVF analysis ..... 19
4.2 Export formats supported by PTA Exporter ..... 20

5.1 Summary of proposed modifications to L-FCPA ..... 27

7.1 Performance metrics for Analysis 1 (Original L-FCPA) ..... 38
7.2 Performance metrics for Analysis 2 (Modified L-FCPA) ..... 38

# Listings


4.1 Input C program for SVF analysis . . . . . 17
6.1 C source code used for pointer analysis . . . . . 30
6.2 Mapping from variables to SLIM operands . . . . . 31
6.3 Pointer to Pointee Relationships . . . . . 31
6.4 Enhanced liveness collection with Ganesan . . . . . 31
6.5 Selective pointer propagation with Ganesan_RHS . . . . . 33
6.6 Pointer Graph Structure Dump . . . . . 35
6.7 Pointer to NodeID Mapping . . . . . 35
A.1 Installing LLVM prerequisites . . . . . 43
A.2 Cloning LLVM repository . . . . . 43
A.3 Configuring LLVM build . . . . . 43
A.4 Building LLVM . . . . . 43
A.5 Installing LLVM . . . . . 43
A.6 Verifying LLVM installation . . . . . 43
C.1 Installing SVF prerequisites . . . . . 47
C.2 Cloning SVF repository . . . . . 47
C.3 Setting LLVM environment variables . . . . . 47
C.4 Building SVF . . . . . 48
C.5 Verifying SVF installation . . . . . 48
C.6 Installing PTA Graphs API prerequisites . . . . . 48
C.7 Cloning PTA Graphs API repository . . . . . 49
C.8 Setting environment variables for PTA Graphs API . . . . . 49
C.9 Sample C program for analysis . . . . . 49
C.10 Running PTA Graphs API analysis . . . . . 49
C.11 Exporting graphs using PTA Exporter . . . . . 50
C.12 Compiling the PTA Exporter program . . . . . 50
C.13 Running the PTA Exporter program . . . . . 50
C.14 Visualizing exported graphs . . . . . 51

# Chapter 1

# Introduction

Program analysis is a vital discipline in software engineering, enabling developers and automated tools to understand, optimize, and verify complex software systems. Among the various techniques in this field, pointer analysis stands out as a cornerstone, particularly for languages like C and C++ that allow direct manipulation of memory through pointers. Pointer analysis determines the set of memory locations a pointer may point to at a program instance during runtime, providing essential information for applications such as compiler optimizations, bug detection, and security audits.

## 1.1 Pointer Analysis

Pointer analysis is a static program analysis technique used in computer science and compiler optimization. Its purpose is to analyze how pointers and memory are manipulated in a program. Pointer analysis identifies the memory locations each pointer may refer to. It can be flow-sensitive or flow-insensitive. Traditional pointer analysis stores the points-to information for all pointers in the program. Flow-sensitive pointer analysis considers the program's control flow, whereas flow-insensitive pointer analysis ignores it.

In Fig. 1.1, variables x and y are of type int*, a and b are of type int. In basic block 1, x points to z, whereas in basic block 2, the points-to information of x is replaced by x points-to a. Now, this new points-to information of x is propagated further to the next basic block.

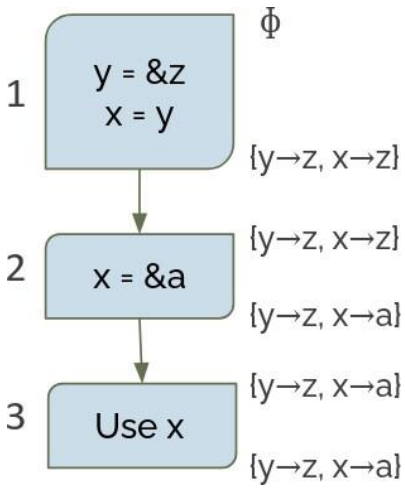



Fig. 1.1: Pointer Analysis

## 1.2 L-FCPA

Traditional pointer analysis computes the points-to information of all pointers throughout the entire program. This gives rise to unnecessary points-to information being computed. In Fig. 1.1 the points-to information of y and x is not needed in basic block 1 and 2. In basic block 3 the points-to information of only x is needed. Hence, traditional pointer analysis is inefficient as it computes lots of points-to pairs. This gave rise to L-FCPA(Liveness-based flow and context-sensitive pointer analysis).

L-FCPA is a bi-directional analysis. In L-FCPA pointer analysis is preceded by a round of liveness analysis. The points-to-information of only those pointers are stored which are live at that particular program instance, therefore eleminating the computation of unnecessary points-to-information. Liveness analysis is a backward analysis and pointer analysis is a forward analysis. In Fig. 1.2 the points-to information of x and y are not stored in the in of the basic blocks 1 and 2 and in the out of the basic block 1 and the points-to information of x are stored in the basic block 2 and in the basic block 3 as they are not needed.

## 1.3 Motivation

L-FCPA works fine when there are no pointer indirections of the form x=*y or *x=y. When there are such indirections L-FCPA needs multiple rounds of liveness

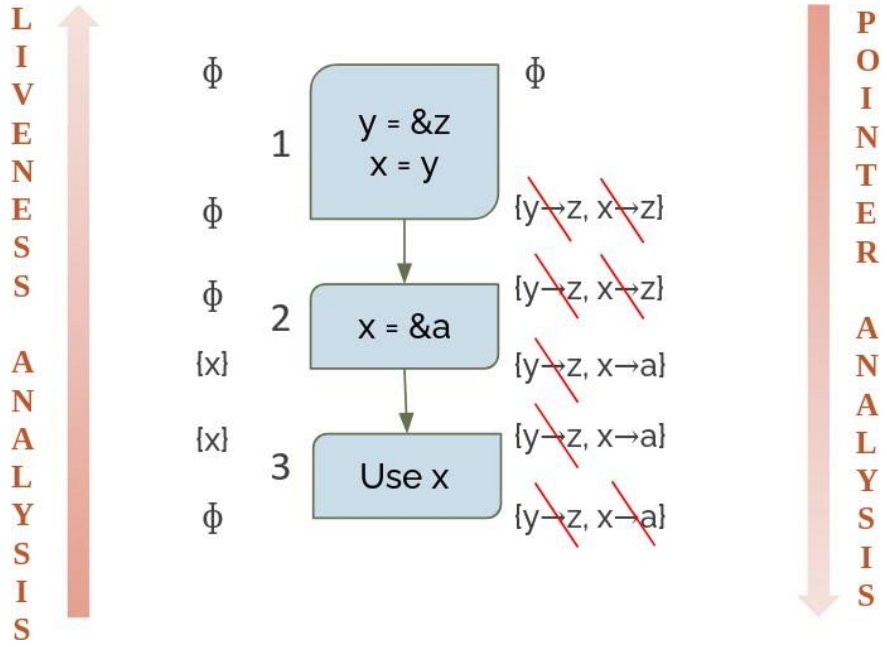



Fig. 1.2: Liveness based pointer analysis

and pointer analysis. In Fig. 1.3 at x=*y, according to L-FCPA the liveness of y

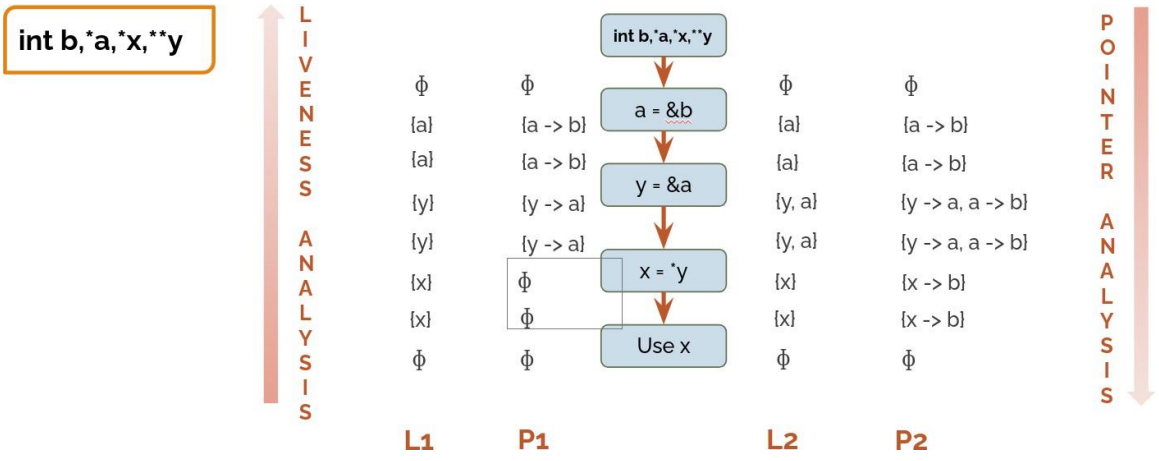



Fig. 1.3: LFCPA with indirection on RHS

along with the pointees of y should be raised in the first round of liveness analysis. But, since the points-to information of y is not available in the first round of liveness analysis L-FCPA generates the liveness of y in this round. Now this is followed by one round of pointer analysis. Since the liveness of y was generated in the previous round of liveness analysis, the points-to information of y is computed in this round of pointer analysis. Now L-FCPA performs another round of liveness analysis followed by a round of pointer analysis. In this second round of liveness

analysis the liveness of pointees of y is generated since the points-to information of y is now available.

## 1.4 Problem Statement

As depicted in the previous section, L-FCPA needs multiple rounds of liveness and pointer analysis when there are pointer indirections. In this thesis, we discuss how to reduce the number of rounds by making a few modifications to the existing L-FCPA.

## 1.5 Thesis Outline

This thesis is organized into nine chapters, each addressing a distinct aspect of the research:

- **Chapter 1: Introduction** – Presents the background of pointer analysis, L-FCPA, motivation and the outline of the thesis.
- **Chapter 2: Literature Review** – Surveys existing pointer analysis techniques, liveness analysis methods, and related tools, establishing the context for our contributions.
- **Chapter 3: L-FCPA** – Details the motivation behind the original L-FCPA framework, examples with pointer indirections, the data flow equations and the L-FCPA algorithm.
- **Chapter 4: SVF** – Here, we take a closer look at the Static Value-Flow Analysis Framework (SVF), a tool that operates on LLVM's intermediate representation (IR). We'll explore what makes SVF tick, how it crunches pointer data, and why it's a key player in our L-FCPA improvements.
- **Chapter 5: Proposed solution** – Explains the original L-FCPA framework, the challenges with pointer indirections, and the proposed solution using weak updates.
- **Chapter 5: LLVM and SLIM** – This chapter gives an outline of SLIM and LLVM intermediate representations.
- **Chapter 6: Implementation Details** – Details the implementation of modifications to the L-FCPA framework. It covers the code adjustments for

handling pointer indirections and the integration of SVF to achieve a more efficient single-round analysis.

- **Chapter 7: Empirical Evaluations** – Focuses on the performance of the modified L-FCPA. This chapter shares the performance results from running our framework on SPEC CPU benchmarks.
- **Chapter 8: Conclusion** – Summarizes the research findings, discusses the limitations of the modified L-FCPA, and proposes directions for future work to further advance pointer analysis techniques.

This structure ensures a comprehensive exploration of the problem, solution, and evaluation, culminating in actionable insights for advancing pointer analysis techniques.

# Chapter 2

# Literature Review

This chapter provides a comprehensive review of the existing research related to pointer analysis, liveness analysis, and the challenges in flow- and context-sensitive analysis. It also discusses existing tools and frameworks, such as SVF, and how they relate to the work presented in this thesis. The literature review is organized into five sections: Pointer Analysis Techniques, Liveness Analysis in Pointer Analysis, Challenges in Flow- and Context-Sensitive Analysis, Existing Tools and Frameworks, and Summary of Related Work.

## 2.1 Pointer Analysis Techniques

Pointer analysis is a static program analysis technique that determines the set of memory locations that a pointer variable may point to at runtime. It is a fundamental analysis used in various applications, such as compiler optimizations, bug detection, and security analysis. The precision and computational complexity of pointer analysis vary depending on the type of analysis employed. The main types of pointer analysis are:

- **Flow-Insensitive Analysis**: This type of analysis ignores the control flow of the program and computes a single points-to set for each pointer variable that holds for the entire program. It is efficient but less precise.
- **Flow-Sensitive Analysis**: This analysis takes into account the control flow of the program and computes different points-to sets for each program point. It is more precise but computationally expensive.
- **Context-Insensitive Analysis**: This analysis does not distinguish between different calling contexts of a function, treating all calls to a function as equivalent.

- **Context-Sensitive Analysis**: This analysis differentiates between different calling contexts, providing more precise results, especially for recursive functions or functions called from multiple sites.

Recent surveys on pointer analysis, such as those by [3] and [8], provide a comprehensive overview of the field and the trade-offs between different analysis techniques.

## 2.2 Liveness Analysis in Pointer Analysis

Liveness analysis is a data flow analysis that determines whether a variable is live at a particular program point, meaning its value may be used along some path from that point before it is redefined. In the context of pointer analysis, liveness information can be used to reduce the computation of points-to sets. By only computing points-to information for live pointers, the analysis can avoid unnecessary computations and improve efficiency.

The Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) proposed by Khedker et al. [4] integrates liveness analysis with pointer analysis to achieve this goal. L-FCPA computes points-to information only for live pointers, using a bidirectional data flow framework that combines liveness and points-to analysis.

### 2.2.1 Challenges in Flow- and Context-Sensitive Analysis

While flow- and context-sensitive analyses provide higher precision, they also introduce significant challenges in terms of scalability and efficiency. The computational complexity of these analyses can be exponential in the size of the program, making them impractical for large codebases.

One specific challenge is the handling of pointer indirections, such as assignments involving dereferences on the left-hand side (LHS) or right-hand side (RHS). In L-FCPA, these indirections create interdependencies between liveness and points-to information, necessitating multiple rounds of analysis to resolve. For example, in the statement $*p = q$, the LHS dereference $*p$ requires knowledge of the pointees of $p$ to determine which variables are affected, but this information is not available until the points-to analysis is performed. This cyclic dependency leads to iterative computations, increasing the analysis time.

Several researchers have discussed these challenges, including Hind [3] and Lhoták [5], who highlight the need for more efficient analysis techniques.

### 2.2.2 Andersen's Flow-Insensitive Pointer Analysis

Andersen's flow-insensitive pointer analysis is a foundational technique in static code analysis, widely used for its balance of scalability and precision in computing points-to sets for pointer variables. Introduced by Lars Ole Andersen in his 1994 Ph.D. thesis [1], this algorithm has become a benchmark for pointer analysis due to its ability to handle large programs efficiently while providing conservative approximations of pointer relationships. Unlike flow-sensitive approaches, which consider the control flow of a program, Andersen's algorithm ignores statement ordering, treating the program as a single node with self-loop edges, as described in [13]. This section explores the mechanics of Andersen's algorithm, its strengths and limitations, and its relevance to our L-FCPA framework.

Mechanics of Andersen's Algorithm Andersen's algorithm, often referred to as an inclusion-based analysis, translates pointer assignments into subset constraints. For a statement of the form $p = q$, the algorithm generates a constraint indicating that the points-to set of $q$ is a subset of the points-to set of $p$ [2]. This process builds a points-to graph iteratively, processing each statement in arbitrary order until the graph stabilizes, i.e., no further changes occur in the points-to sets [10]. The algorithm operates in three phases: constraint generation, constraint optimization, and constraint solving. In the first phase, it translates LLVM IR into a set of constraints; in the second, it applies optimizations like HVN and HU to merge pointers with equivalent points-to sets; and in the third, it uses techniques like HCD and LCD to detect strongly connected components in the constraint graph, reducing computational overhead [2].

## 2.3 SVF

Several tools and frameworks have been developed to perform pointer analysis, including the Static Value-Flow Analysis Framework (SVF) [11]. SVF is an open-source tool built on top of LLVM that provides various types of pointer analysis, including flow-sensitive, context-sensitive, and flow-insensitive analyses.

SVF is particularly relevant to our work as it can compute flow-insensitive points-to sets efficiently. These points-to sets can be used to approximate the possible targets of pointers without considering the program's control flow, providing a fast and conservative analysis. In our research, we leverage SVF's flow-insensitive analysis to optimize the handling of RHS indirections in L-FCPA, reducing the need for multiple analysis rounds.

## 2.4 Summary of Related Work

The literature on pointer analysis is extensive, with numerous techniques and tools developed to address the trade-offs between precision and efficiency. While flow- and context-sensitive analyses offer high precision, their scalability remains a significant challenge, particularly for large programs. Liveness-based approaches, such as L-FCPA, have shown promise in improving efficiency by reducing unnecessary computations. However, the handling of pointer indirections in these analyses still requires multiple rounds, limiting their scalability.

Our work aims to address this gap by proposing modifications to L-FCPA that reduce the number of analysis rounds needed for pointer indirections, leveraging weak updates and flow-insensitive points-to sets from SVF.

# Chapter 3

# L-FCPA - Liveness Based Flow and Context Sensitive Pointer Analysis

## 3.1 L-FCPA

Traditional pointer analysis computes the points-to information of all pointers throughout the entire program. This gives rise to unnecessary points-to information being computed. In Fig. 1.1 the point-to-information of y and x is not needed in basic block 1 and 2. In basic block 3 the point-to information of only x is needed. Hence, traditional pointer analysis is inefficient.

L-FCPA is a bi-directional analysis. In L-FCPA pointer analysis is preceded by a round of liveness analysis. The points-to-information of only those pointers are stored which are live at that particular program instance, therefore eleminating the computation of unnecessary points-to-information. Liveness analysis is done starting from the last basic block in a bottom up manner and pointer analysis in a top down manner. In Fig. 1.2 the information points of x and y are not stored in the in of the basic blocks 1 and 2 and in the out of the basic block 1 and the information points of x are stored in the basic block 2 and in the basic block 3.

## 3.2 L-FCPA when there are indirections on the LHS

When there are pointer indirections of the form *x=y or x=*y, L-FCPA needs multiple rounds of liveness and pointer analysis. In Fig. 3.1 at *z=x, the points to information of z is not available in the first round of liveness analysis. Hence L-FCPA blocks the propagation of all the variables that are live at out of that instruction and generates the liveness of z. In the round of pointer analysis the points to information of z is computed as the liveness of z has been generated in

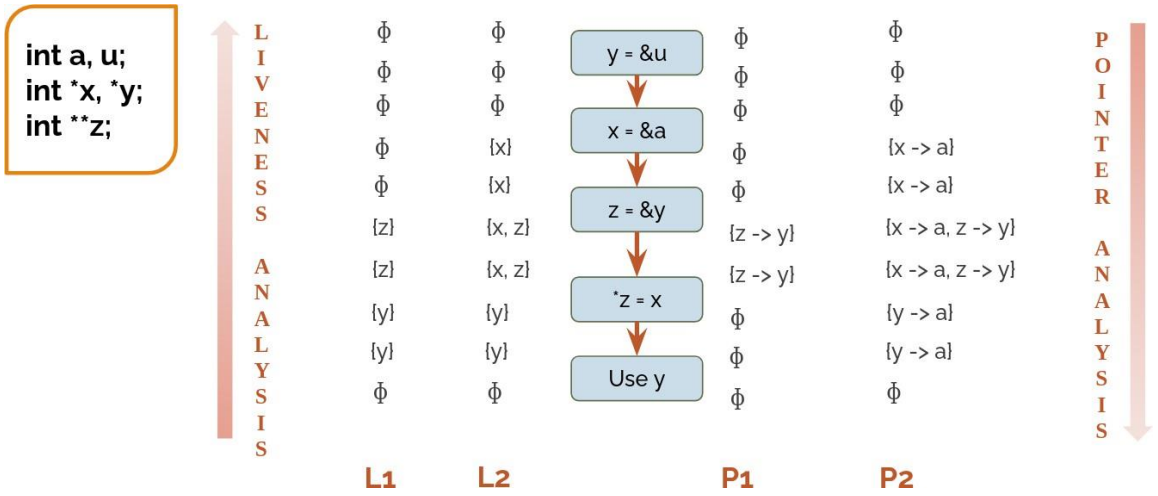



Fig. 3.1: LFCPA with indirection on LHS

the previous round of liveness analysis. Now another round of liveness analysis is performed in which the liveness of pointees of z are killed if it is live at the out of this instruction and the liveness of x is generated, followed by another round of pointer analysis.

## 3.3 L-FCPA Equations

The Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework, introduced by [4], employs a bidirectional data flow approach to compute precise points-to information for live pointers. This section elucidates the core data flow equations governing liveness and points-to analysis, as presented in the original paper. These equations underpin the framework's ability to integrate liveness analysis with pointer analysis, reducing computational overhead while maintaining flow and context sensitivity. We also discuss the role of extractor functions and illustrate the data flow process with a diagram.

### 3.3.1 Liveness Analysis Equations

Liveness analysis in L-FCPA determines which pointer variables are live at each program point, meaning their values may be used before being redefined. The liveness information is represented by data flow variables $\mathrm{Lin}_{?}$ and $\mathrm{Lout}_{?}$, associated with the entry and exit of a control flow graph (CFG) node ?, respectively. The lattice for liveness is defined as $L = \langle P(\boldsymbol{P}), \supseteq \rangle$, where $\boldsymbol{P} \subset \boldsymbol{V}$ is the set of pointer variables, and $P(\boldsymbol{P})$ denotes the power set of $\boldsymbol{P}$.

The liveness equations are formulated as follows:

$$\text{Lin}_n = (\text{Lout}_n - \text{Kill}_n) \cup \text{Ref}_n \tag{3.1}$$

$$\text{Lout}_n = \bigcup_{s \in \text{succ}(n)} \text{Lin}_s \tag{3.2}$$

Here, $\text{Kill}_n$ represents the set of pointer variables definitely modified by node $n$, and $\text{Ref}_n$ denotes the variables that become live due to their use in $n$. The $\text{Kill}_n$ set is influenced by the points-to information ($\text{Ain}_n$) through the Must function, particularly for statements like $*x = y$, where the pointees of $x$ determine the killed variables. The boundary condition sets $\text{Lout}_{e_p} = \emptyset$ for the exit node $e_p$ of a procedure $p$, reflecting that no pointers are live upon procedure exit.

These equations ensure that liveness propagates backward through the CFG, capturing only the pointers relevant to subsequent computations [4].

### 3.3.2 Points-To Analysis Equations

Points-to analysis computes the set of memory locations that each live pointer may reference, represented by $\text{Ain}_n$ and $\text{Aout}_n$ for the entry and exit of node $n$. The lattice for points-to information is $A = \langle P(\boldsymbol{P} \times \boldsymbol{V}), \supseteq \rangle$, where $\boldsymbol{V}$ is the set of all variables, and $\boldsymbol{P} \times \boldsymbol{V}$ represents pointer-location pairs.

The points-to equations are:

$$\text{Ain}_n = \bigcap_{p \in \text{pred}(n)} \text{Aout}_p \tag{3.3}$$

$$\text{Aout}_n = (\text{Ain}_n - \{(x, t) \mid x \in \text{Def}_n\}) \cup (\text{Def}_n \times \text{Pointee}_n) \tag{3.4}$$

In Equation 3.3, $\text{Ain}_n$ is the meet (intersection) of points-to sets from predecessor nodes, ensuring flow sensitivity. Equation 3.4 updates the points-to set by removing pairs for modified pointers ($\text{Def}_n$) and adding new pairs based on the pointees assigned ($\text{Pointee}_n$). The boundary condition sets $\text{Ain}_{e_p} = \text{Lin}_{e_p} \times \{?\}$, where ? denotes an undefined location, initializing pointers with invalid addresses at the procedure entry $e_p$.

### 3.3.3 Extractor Functions

The equations rely on extractor functions to process pointer-related data:

- Def□: Identifies pointer variables modified by node □. - Kill□: Determines variables definitely modified, influenced by the Must function for indirections. - Ref□: Computes variables that become live due to use in □, adjusted for strong liveness. - Pointee□: Identifies potential pointees assigned in □. - Must(□): Extracts must-points-to information from may-points-to sets □, defined as:

$$\text{Must}(□) = \begin{cases} \bigcup_{□ \in P} \{□\} \times \{□\} & \text{if } □\{□\} = \emptyset \vee □\{□\} = \{?\} \\ \emptyset & \text{if } □\{□\} = \{□\} \wedge □ \neq ? \\ \emptyset & \text{otherwise} \end{cases}$$

These functions bridge liveness and points-to analyses, enabling precise handling of pointer assignments [4].

### 3.3.4 Data Flow Framework

The interplay of liveness and points-to equations forms a bidirectional data flow framework, as shown in Fig. 3.2. The diagram illustrates how liveness propagates backward and points-to information flows forward, with extractor functions mediating their interaction.

| Extractor functions for statement $n$ ($Def_n, Kill_n, Ref_n \subseteq \boldsymbol{P}$; $Pointee_n \subseteq \boldsymbol{V}$)<br>Notation: we assume that $x, y \in \boldsymbol{P}$ and $a \in \boldsymbol{V}$. $A$ abbreviates $Ain_n$. | | | | | |
|---|---|---|---|---|---|
| Stmt. | $Def_n$ | $Kill_n$ | $Ref_n$ | | $Pointee_n$ |
| | | | if $Def_n \cap Lout_n \neq \emptyset$ | Otherwise | |
| *use* $x$ | $\emptyset$ | $\emptyset$ | $\{x\}$ | $\{x\}$ | $\emptyset$ |
| $x = \&a$ | $\{x\}$ | $\{x\}$ | $\emptyset$ | $\emptyset$ | $\{a\}$ |
| $x = y$ | $\{x\}$ | $\{x\}$ | $\{y\}$ | $\emptyset$ | $A\{y\}$ |
| $x = *y$ | $\{x\}$ | $\{x\}$ | $\{y\} \cup (A\{y\} \cap \boldsymbol{P})$ | $\emptyset$ | $A(A\{y\} \cap \boldsymbol{P})$ |
| $*x = y$ | $A\{x\} \cap \boldsymbol{P}$ | $Must(A)\{x\} \cap \boldsymbol{P}$ | $\{x, y\}$ | $\{x\}$ | $A\{y\}$ |
| other | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

Fig. 3.2: Data flow framework of L-FCPA
Liveness and points-to analysis interaction

This framework ensures that only live pointers are analyzed, reducing compu-

tational overhead while maintaining precision [5].

# Chapter 4

# SVF - Static Value Flow Analysis Framework

The Static Value-Flow Analysis Framework (SVF) is a pivotal tool in static program analysis, particularly for pointer analysis tasks essential to this research. This chapter provides an in-depth exploration of SVF, focusing on its role in generating flow-insensitive points-to sets for the modified Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework. The chapter is structured as follows: Section 4.1 introduces SVF, Section 4.2 presents a sample input-output example with commands to run SVF analysis, and Section **??** is reserved for the PTA Exporter, to be detailed later with provided information.

## 4.1 Introduction to SVF

SVF is an open-source static analysis framework designed for analyzing LLVM-based languages, such as C and C++, by leveraging LLVM's intermediate representation (IR) [11, 7]. Developed to support pointer analysis and value-flow analysis, SVF is widely used for tasks like bug detection, security analysis, and program optimization. Its modular design allows for various analysis types, including flow-sensitive, context-sensitive, and flow-insensitive pointer analyses, making it a versatile tool for integrating with frameworks like L-FCPA.

SVF operates on LLVM IR (bitcode), constructing graphs (e.g., control-flow graphs, call graphs, constraint graphs) to model program behavior. For this research, we utilize SVF's flow-insensitive pointer analysis, specifically Andersen's algorithm, to precompute points-to sets for RHS indirections, as detailed in Chapter 3 [1]. SVF's repository provides robust documentation and source code, facilitating its integration into our modified L-FCPA framework [12].

The framework's architecture comprises three core components: *Graph*, *Rules*, and *Solver*. The Graph component builds representations like the constraint graph, Rules define pointer assignment logic, and the Solver resolves constraints iteratively [11]. This structure, illustrated in Fig. 4.1, enables efficient analysis, crucial for our single-round L-FCPA approach.

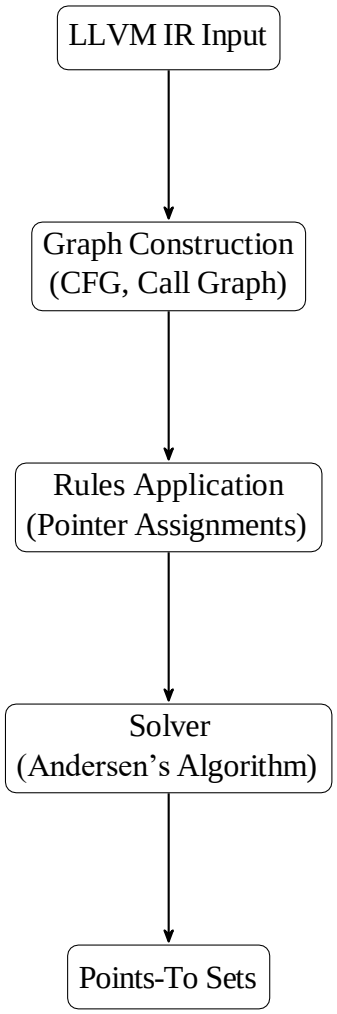



Fig. 4.1: SVF analysis workflow
From LLVM IR to points-to sets

## 4.2 SVF Input-Output Example

The Static Value-Flow Analysis Framework (SVF) is employed to perform pointer analysis on the thesis codebase using the Liveness-Based Flow- and Context-Sensitive Pointer Analysis (LFCPA) variant [11]. This section examines an example C program, its analysis by SVF, and the resulting points-to information, highlighting how pointer variables propagate and interact with memory locations at compile time. The output provides critical insights into pointer relationships,

aiding optimization and safety verification in our L-FCPA framework [4].

### 4.2.1 Input Program

The input program, defined in the file Thesis_example.c, manipulates pointers to integers:

```c
int a, b;
int *x, *y;

int main()
{
    a = 10;
    b = 20;
    x = &a;
    y = &b;
    x = y;
    return 0;
}
```

Listing 4.1: Input C program for SVF analysis

In Listing 4.1, variables a and b are initialized to 10 and 20, respectively. Pointers x and y are assigned the addresses of a and b, and then x is updated to point to the same location as y, which is b.

### 4.2.2 SVF Analysis and Output

SVF performs a flow-insensitive pointer analysis on this program, mapping pointer identifiers (Ptr IDs) to abstract memory objects or variables (Target Node IDs) they may point to. Each pointer is associated with its declaration location in Thesis_example.c, specified by line number (ln) and file name (fl). The output, visualized as a points-to graph in Fig. 4.2, reveals the following key observations [11]:

- **Uninitialized Pointers**: Ptr 6 and Ptr 8, both declared at line 2, have empty points-to sets (PointsTo = *{}*), indicating they do not point to any statically known memory locations. Ptr 6 corresponds to variable a (unresolved), and Ptr 8 is the target node for b, suggesting either uninitialized pointers or incomplete alias information at this stage.
- **Direct Points-to Relationships**:
  - Ptr 7, declared at line 2, points to Target Node 8 (PointsTo = *{8}*), which corresponds to the memory location of variable b.

  - Ptr 9, declared at line 3, points to Target Node 10 (PointsTo = {10}), mapping to variable x.
  - Ptr 11, declared at line 3, points to Target Node 12 (PointsTo = {12}), representing variable y.

- **Aliasing and Multiple Points-to**:
  - Ptr 10 points to both Target Node 6 and Target Node 8 (PointsTo = {6, 8}), indicating that x may alias with variables a and b. This reflects the reassignment in line 5 (x = y), where x initially points to a (line 3) but is updated to point to b via y.
  - Ptr 12 points to Target Node 8 (PointsTo = {8}), confirming that y points to b, and highlighting that multiple pointers (x and y) refer to the same memory location (b), suggesting aliasing.

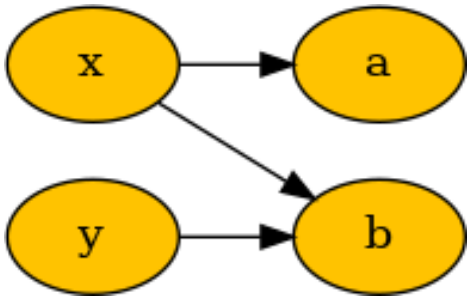



Fig. 4.2: Simplified SVF points-to graph for the input program
Showing pointer relationships

### 4.2.3 Points-to Information Analysis

The points-to graph in Fig. 4.2 visualizes the relationships between pointers and memory locations, providing a clear mapping of pointer interactions:

- **Unresolved Pointers**: The empty points-to sets for Ptr 6 and Ptr 8 indicate unresolved or uninitialized pointers at this analysis stage. For Ptr 6 (likely a), this suggests no pointer directly references a after the reassignment of x. For Ptr 8, as the target node for b, the empty set may reflect an intermediate analysis state before all relationships are resolved.

- **Aliasing Patterns**: Ptr 10 (x) pointing to both a and b (Target Nodes 6 and 8) shows aliasing due to the reassignment x = y. Ptr 12 (y) and Ptr 10 both pointing to Target Node 8 (b) confirm that x and y share the same memory

location after the assignment, a common aliasing pattern in pointer-heavy code.

- **Direct Mappings**: Ptr 7, Ptr 9, and Ptr 11 map directly to Target Nodes 8, 10, and 12, respectively, reflecting straightforward pointer assignments (`y = &b`, `x`'s node, `y`'s node).

Table 4.1 summarizes these pointer relationships, detailing each pointer's declaration line, points-to set, and corresponding variable.

| Pointer ID | Declared Line | Points To (Target Node ID) | Likely Variable |
|---|---|---|---|
| Ptr 6 | 2 | — | `a` (unresolved) |
| Ptr 7 | 2 | 8 | `b` |
| Ptr 8 | 2 | — | `b` (target) |
| Ptr 9 | 3 | 10 | `x` |
| Ptr 10 | 3 | 6, 8 | `a`, `b` |
| Ptr 11 | 3 | 12 | `y` |
| Ptr 12 | 3 | 8 | `b` |

Table 4.1: Summary of pointer relationships from SVF analysis

## 4.3 PTA Exporter

The PTA Exporter module consists of nodes (representing memory objects or pointer variables) and edges (indicating points-to relations), and exports it into various formats suitable for different use cases. Each node in the graph, as defined by the `PtrDataT` structure, contains metadata such as the variable name (`var`), source file (`file`), and line number (`line`), ensuring traceability to the original source code. The edges, stored as pairs of `NodeIdT` values in a `std::vector<std::pair<NodeIdT, NodeIdT>>`, represent direct or indirect points-to relationships.

The exported data can be used for:

- **Visualization**: Generating graphical representations of pointer relationships for debugging or presentation.
- **Analysis**: Enabling external tools to perform further static analyses, such as liveness or memory leak detection.
- **Integration**: Supporting integration with other systems, such as optimization frameworks or safety verification tools.

| Format | Description | Use Case |
|---|---|---|
| DOT | Exports the graph as a DOT file compatible with Graphviz, representing nodes and edges for visualization. | Visualizing pointer relationships in tools like Graphviz. |
| JSON | Outputs the graph as a JSON object, with nodes and edges serialized for programmatic access. | Integration with external analysis tools or scripts. |
| CSV | Generates a comma-separated file listing nodes (PtrDataT) and edges (NodeIdT pairs). | Data export for spreadsheet analysis or custom processing. |

Table 4.2: Export formats supported by PTA Exporter

### 4.3.1 PTA Exporter API

The PTA Exporter APIs defined in the header file, includes export functionalities while maintaining a minimal and expressive interface. Key methods include:

- exportToDOT(const std::string& filename): Exports the graph to a DOT file for visualization with Graphviz.
- exportToJSON(const std::string& filename): Serializes the graph into a JSON file for integration with external tools.
- exportToCSV(const std::string& filename): Writes nodes and edges to a CSV file for data analysis.

These methods complement the existing PTA Explorer API (e.g., getPointees, addEdge), abstracting the complexity of SVF's internal representation and enabling seamless export of pointer analysis results [11].

## 4.4 Changes Done to the Header File Format

The original header file defined a data structure called node_map of type std::map¡PtrDataT, NodeIdT¿, which maps node metadata to its corresponding node ID. The PtrDataT structure contains metadata including the variable name (var), the source file (file), and the line number (line) where the variable is declared. However, in the Static Value-Flow Analysis Framework (SVF), some nodes lack this information [11]. In such cases, we store empty strings for var, file, and line,

leading to multiple entries with identical PtrDataT values (i.e., all three fields as empty strings). Since std::map does not allow duplicate keys, this caused issues with storing all node details. To accommodate the details of all nodes, including duplicates, we changed the data type of node map from std::map to std::vector<std::pair<PtrDataT, NodeIdT>>.

# Chapter 5

# Proposed Solution

This chapter outlines the proposed modifications to the Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework to eliminate the need for multiple analysis rounds caused by pointer indirections. By introducing a weak update mechanism for left-hand side (LHS) indirections and integrating flow-insensitive points-to sets from the Static Value-Flow Analysis Framework (SVF), we achieve a single-round analysis while preserving precision. The chapter is structured as follows: Section 5.1 provides an overview, Section 5.2 details the weak update for LHS indirections, Section 5.3 describes SVF integration for RHS indirections, Section 5.6 presents the single-round framework, and Section 5.7 summarizes the modifications with a table of equations.

## 5.1 Motivation

The L-FCPA framework, developed by [4], is an efficient approach to pointer analysis that computes points-to information only for live pointers. However, its handling of pointer indirections, such as $*x = y$ (LHS) and $x = *y$ (RHS), requires iterative rounds due to dependencies between liveness and points-to computations [3]. Our solution addresses this by:

- Deciding strong or weak update for LHS indirections to compute liveness in a single pass.
- Leveraging SVF's flow-insensitive points-to sets to precompute pointees for RHS indirections, available at [12] [11].

These modifications enhance L-FCPA's scalability, making it suitable for large-scale program analysis.

## 5.2 Weak Update for LHS Indirections

As discussed in 3.2 in the original L-FCPA, LHS indirections (*x = y) block the propagation of live variables at the statement's output, marking only x as live and awaiting points-to analysis to resolve x's pointees. This necessitates additional liveness rounds [4].

We propose a weak update mechanism that includes x, y, and all live variables at the output, eliminating iteration.

The modified transfer function is:

$$Lin_{\square} = (Lout_{\square}) \cup Ref_{\square} \cup \{\square, \square\} \tag{5.1}$$

By setting $Kill_{\square} = \emptyset$, this ensures all potentially affected variables remain live, completing liveness analysis in one pass [5].

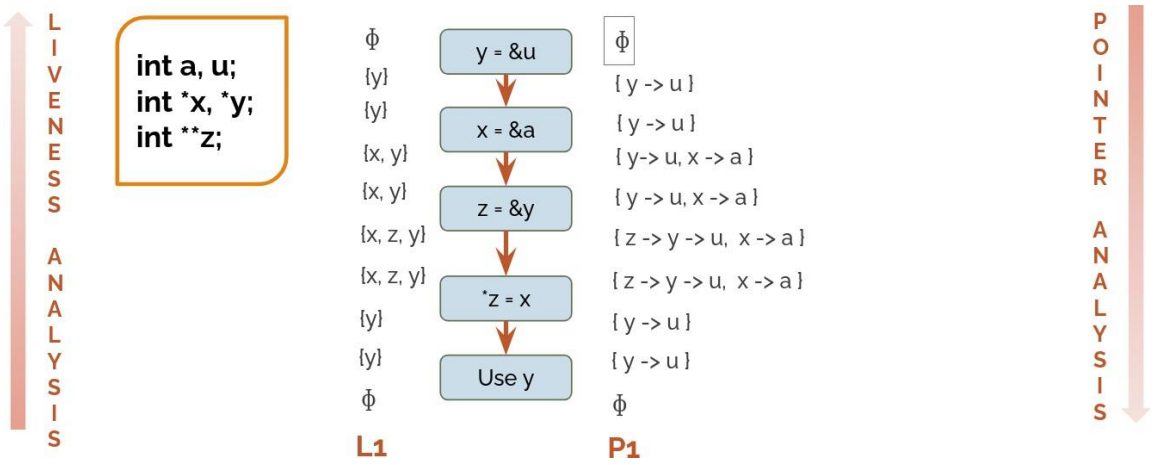



Fig. 5.1: Modified L-FCPA by doing weak update

When there is a pointer indirection on LHS, instead of blocking the propagation of all the live variables at the out of this instruction we perform a weak update by generating the liveness of left-hand side variable, right-hand side variable and the live variables at the out of that instruction. In Fig. 5.1 at the instruction *z=x we raise the liveness of z, x and y. By doing so, all the points to information that are needed will be computed in the next round of pointer analysis itself and hence another extra round of liveness followed by pointer analysis is not needed.

## 5.3 SVF Integration for RHS Indirections

As discussed previously in 1.3 for RHS indirections (x = *y), the original L-FCPA generates only y's liveness initially, as y's pointees are resolved later, triggering further rounds. We use SVF's flow-insensitive points-to sets, generated via Andersen's algorithm, to precompute pt(□). SVF provides efficient pointer analysis [11, 12]. This enables liveness analysis to account for all pointees in a single pass, decoupling it from iterative points-to computations.

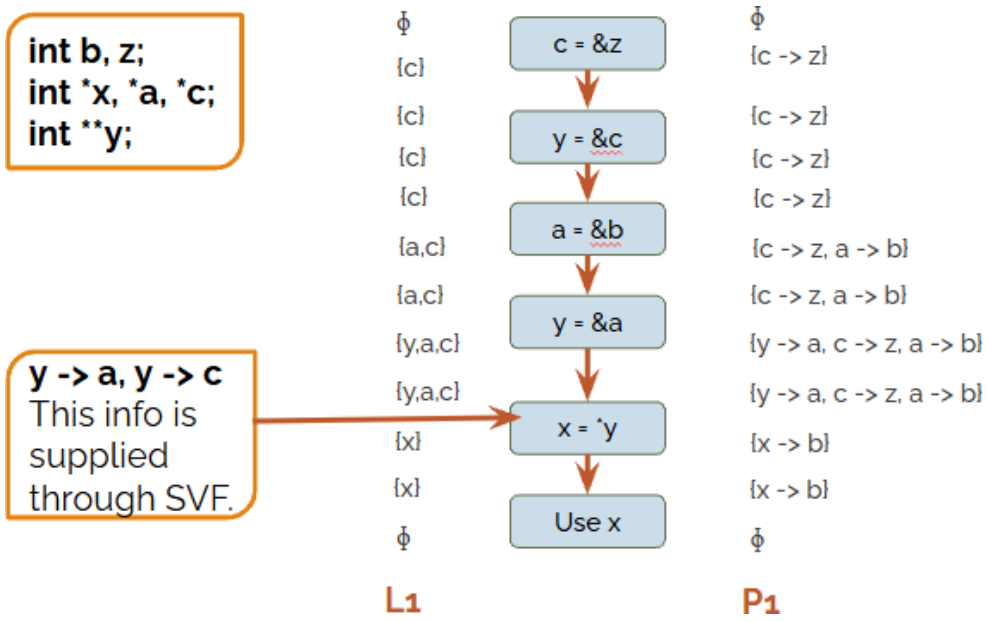



Fig. 5.2: Example illustrating modified L-FCPA for RHS indirections

Statement x = *y

Figure 5.2 illustrates the modified L-FCPA's handling of RHS indirections, exemplified by the statement x = *y. In this approach, points-to information for y is provided by the Static Value-Flow Analysis Framework (SVF) [11, 12]. Consequently, during the initial liveness analysis round, the liveness of y and its pointees is generated using SVF's flow-insensitive points-to sets. This is followed by a single round of pointer analysis, where the points-to information for y and its pointees is computed, as their liveness has already been established [4].

## 5.4 Handling Function Pointers with SVF Data

Function pointers introduce additional complexity in interprocedural pointer analysis, as their pointees (callee functions) must be resolved to accurately propagate liveness across function calls. In the original L-FCPA, resolving function pointer

targets often requires multiple rounds, as the pointees of function pointers are not known until points-to analysis completes [4]. To address this, we propose using SVF's flow-insensitive points-to sets to precompute the pointees of function pointers, raising their liveness in a single pass.

For a function pointer fp, we query SVF to obtain its points-to set pt(fp), which contains the possible callee functions. We then include these callees in the liveness set, ensuring their variables are marked live for interprocedural analysis.

## 5.5 Modified L-FCPA Equations

This section elucidates the original data flow equations of the Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework for handling pointer indirections, as proposed by Khedker et al. [4], and presents the modified equations introduced in this research to address their inefficiencies. The original equations require multiple rounds due to cyclic dependencies between liveness and points-to computations for left-hand side (LHS) and right-hand side (RHS) indirections. Our modifications, utilizing a weak update mechanism for LHS indirections and flow-insensitive points-to sets from the Static Value-Flow Analysis Framework (SVF) for RHS indirections, enable a single-round analysis. The section compares these equations, includes a table summarizing the changes, and illustrates the modified data flow process.

### 5.5.1 Original L-FCPA Equations for Indirections

In L-FCPA, liveness and points-to analyses are interdependent, with equations defined over a control flow graph (CFG) node $n$. Liveness is represented by $\text{Lin}_n$ and $\text{Lout}_n$ (entry and exit liveness sets), and points-to information by $\text{Ain}_n$ and $\text{Aout}_n$. The lattices are $L = \langle P(\boldsymbol{P}), \supseteq \rangle$ for liveness, where $\boldsymbol{P} \subset \boldsymbol{V}$ is the set of pointer variables, and $A = \langle P(\boldsymbol{P} \times \boldsymbol{V}), \supseteq \rangle$ for points-to sets, where $\boldsymbol{V}$ is the set of all variables [4].

For LHS indirections ($*x = y$), the liveness equation is:

$$\text{Lin}_n = (\text{Lout}_n - \text{Kill}_n) \cup \text{Ref}_n \tag{5.2}$$

Here, $\text{Kill}_n$ depends on $\text{Ain}_n$ via the Must function, which identifies pointers definitely modified (e.g., pointees of $x$). $\text{Ref}_n$ includes $x$ as live, regardless of its pointees' liveness, to ensure pointees are discovered. The points-to equation updates $\text{Aout}_n$ based on $\text{Def}_n$ (modified pointers) and $\text{Pointee}_n$ (assigned pointees):

$$Aout_{□} = (Ain_{□} - \{(□, □) \mid □ \in Def_{□}\}) \cup (Def_{□} \times Pointee_{□}) \quad (5.3)$$

For RHS indirections ($x = *y$), the same liveness equation applies, but $Ref_{□}$ includes $y$, and the liveness of $y$'s pointees depends on subsequent points-to analysis, triggering additional rounds.

These equations, while precise, require iteration because $Kill_{□}$ and $Ref_{□}$ depend on points-to sets, which are computed after liveness [3].

### 5.5.2 Modified Equations for LHS Indirections

To eliminate multiple rounds for LHS indirections, we propose a weak update mechanism. Instead of awaiting points-to analysis to resolve $x$'s pointees, we conservatively include $x$, $y$, and all live variables at the output in the liveness set. The modified liveness transfer function is:

$$Lin_{□} = (Lout_{□}) \cup Ref_{□} \cup \{□, □\} \quad (5.4)$$

By setting $Kill_{□} = \emptyset$, this ensures no variables are excluded prematurely, allowing liveness analysis to complete in one pass. The points-to equation remains unchanged (Equation 5.3), as the weak update affects only liveness [5].

### 5.5.3 Modified Equations for RHS Indirections

For RHS indirections, we integrate flow-insensitive points-to sets from SVF [11, 12]. SVF's Andersen's algorithm precomputes pt(□), the pointees of $y$, enabling their inclusion in the initial liveness round. The modified liveness transfer function is:

$$Lin_{□} = (Lout_{□} - Kill_{□}) \cup Ref_{□} \cup \{□\} \cup pt(□) \quad (5.5)$$

This approach decouples liveness from iterative points-to computations, as pt(□) is available upfront. The points-to equation (Equation 5.3) is applied subsequently in a single pass.

## 5.6 Single-Round Analysis Framework

The proposed modifications create a single-round analysis workflow, supported by LLVM's infrastructure [6]. The process is:

1. **Precompute Points-To Sets**: Use SVF to generate flow-insensitive points-to sets.
2. **Liveness Analysis**: Apply Equation 5.4 for LHS indirections and Equation 5.5 for RHS indirections in one pass.
3. **Points-To Analysis**: Compute precise points-to sets for live pointers in one pass.

## 5.7 Summary of Proposed Modifications

Table 5.1 summarizes the proposed modifications, their objectives, equations, and associated tools, providing a concise reference.

| Modification | Objective | Equation/Mechanism | Tool/Link |
|---|---|---|---|
| Weak Update for LHS Indirections | Single-pass liveness for $*x = y$ | $Lin_{\square} = (Lout_{\square} \setminus \emptyset) \cup Ref_{\square} \cup \{\square, \square\}$ | LLVM [7] |
| SVF Integration for RHS Indirections | Precompute pointees for $x = *y$ | $Lin_{\square} = (Lout_{\square} \setminus Kill_{\square}) \cup Ref_{\square} \cup \{\square\} \cup pt(\square)$ | SVF [11] |
| Function Pointer Handling with SVF Data | Raise liveness of function pointer pointees | $Lin_{\square} = (Lout_{\square} \setminus Kill_{\square}) \cup Ref_{\square} \cup \{\square\square\} \cup pt(\square\square)$ | SVF [11] |
| Single-Round Workflow | Eliminate iterative analysis | Precompute points-to sets, single-pass liveness, single-pass points-to | LLVM, SVF |

Table 5.1: Summary of proposed modifications to L-FCPA

# Chapter 6

# Implementation Details

This chapter outlines the implementation of the modified Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework, focusing on optimizations to reduce the number of analysis rounds caused by pointer indirections. The modifications enhance liveness propagation and improve the handling of pointer indirections, using custom preprocessor macros to isolate experimental behavior from the default implementation. The chapter is structured as follows: Section 6.1 outlines the flow of the program, Section 6.2 illustrates the intialization of the variable mappings and points-to data, Section 6.3 describes the enhanced liveness collection for incoming operands, Section 6.4 details the improved points-to propagation for pointer indirections, Section 6.5 describes the modifications incase function pointers and Section 6.6 outlines the changes done to PTA Exporter.

## 6.1 Architecture

Figure 6.1 illustrates the overall workflow of our pointer and liveness analysis system that integrates LLVM, SVF, and the L-FCPA framework. The architecture is designed to process a C input file, extract pointer information using SVF, and perform liveness analysis using a modified version of L-FCPA.

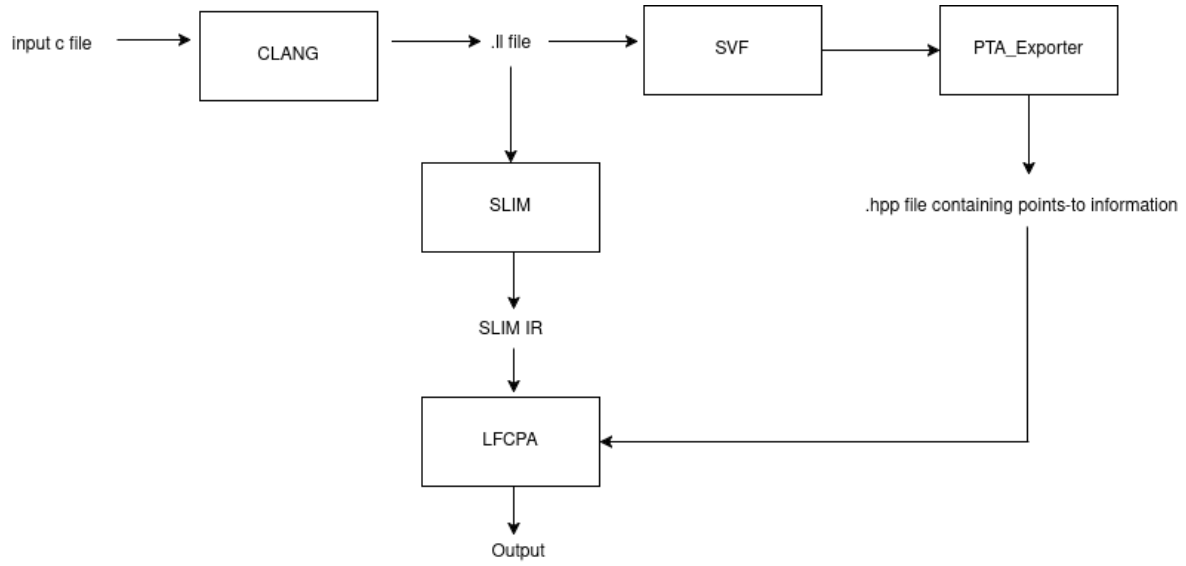



Fig. 6.1: Flow chart depicting the flow of program

### 6.1.1 Step 1: Generating LLVM IR from C Source Code

The input to the system is a standard C source file. This file is compiled using the clang front-end with LLVM flags enabled to produce a human-readable LLVM IR file (.ll). The IR preserves variable-level information and is compatible with downstream analysis tools such as SVF.

### 6.1.2 Step 2: Extracting Points-To Information with SVF

The generated LLVM IR file is then passed to the **PTA Exporter** built over the SVF (Static Value-Flow) framework. The PTA Exporter analyzes the IR to compute the pointer aliasing (points-to) relationships. These relationships are then exported into a C++ header file (e.g., pta_graphs.hpp) using graph representations such as pointer graphs and call graphs.

### 6.1.3 Step 3: Feeding Points-To Information into L-FCPA

The pointsToGraph.hpp file is included in the L-FCPA implementation. The L-FCPA framework reads the points-to information and combines it with data-flow logic to perform liveness analysis in the presence of pointer indirections. This integration enables more precise and scalable pointer-aware program analysis.

### 6.1.4 Step 4: Producing Analysis Output

The final output of the system includes enhanced liveness sets and points-to maps, which are used for debugging, optimization, and correctness verification of the target program. This modular flow also allows for testing improvements in either the SVF layer or the L-FCPA logic independently.

## 6.2 Initialization of Variable Mapping and Points-To Data

Before performing the L-FCPA analysis, we initialize the necessary data structures to map variables to their SLIM operands and populate the points-to information from SVF. This initialization ensures that we can efficiently handle indirections on the LHS, RHS, and function pointers during the analysis. The process involves the following steps:

### 6.2.1 Extracting and Storing Variable Information

When our L-FCPA pass begins running on LLVM, we execute the function `extractAndStoreVariableInfo`, which takes an `llvm::Module` as an argument. This function stores the variable names and line numbers of all global variables, along with their corresponding SLIM operands of the form `SLIMOperand*` in a data structure named `variableToSlimOperandMap`, which is of type `std::map <std::pair<std::string, std::string>, SLIMOperand*>`. Subsequently, we iterate through all functions and basic blocks in the module, storing the local variable names, function names, and their line numbers, along with their SLIM operands, in `variableToSlimOperandMap`. This mapping enables efficient retrieval of SLIM operands based on variable metadata [7].

```
int a, b;
int *x, *y;

int main()
{
    a = 10;
    b = 20;
    x = &a;
    y = &b;
    x = y;
    return 0;
}
```

Listing 6.1: C source code used for pointer analysis

```
-------variableToSlimOperandMap Contents -------
Variable: a, Line: 1 -> a
Variable: b, Line: 1 -> b
Variable: i, Line: 10 -> i_main
Variable: main, Line: 4 -> nullptr
Variable: x, Line: 2 -> x
Variable: y, Line: 2 -> y
```

Listing 6.2: Mapping from variables to SLIM operands

### 6.2.2 Populating the Points-To Set

Next, we execute the function populatePointeeSet, which takes optIR and a pointer to the module as arguments. The SVF output, stored in pointsToGraph.hpp, contains points-to information in the form of variable names and line numbers. Using variableToSlimOperandMap, the corresponding SLIM operands can be fetched for all variables in SVF. This points-to information is then populated in pointeeSet, which is of the form std::map<SLIMOperand*, std::set<SLIMOperand*>>. This data structure maps each SLIM operand to a set of SLIM operands representing its pointees [11].

```
------PointeeSet Contents------
Pointer: b -> Pointees: b,
Pointer: x -> Pointees: b, a,
Pointer: y -> Pointees: b,
Pointer: a -> Pointees: a,
```

Listing 6.3: Pointer to Pointee Relationships

### 6.2.3 Performing L-FCPA Analysis

With the pointeeSet data prepared, we are ready to perform the L-FCPA analysis, handling indirections on the LHS, RHS, and function pointers using the precomputed points-to information. This initialization ensures that the analysis can proceed efficiently in a single round, as described in subsequent sections.

## 6.3 Enhanced Liveness Collection for Incoming Operands

```
std::set<SLIMOperand *> INofInst1;
INofInst1.insert(source);
```

```
for (auto out : OUTofInst) {
    INofInst1.insert(out);
}
std::vector<std::pair<SLIMOperand *, int>> RHS = I->getRHS();
for (const auto &rhsPair : RHS)
{
  SLIMOperand *rhsOperand = rhsPair.first;
  INofInst.insert(rhsOperand);
}
return INofInst1;
```

Listing 6.4: Enhanced liveness collection with Ganesan

To address the inefficiency of multiple liveness rounds in the original L-FCPA, particularly for instructions involving pointer indirections, we introduced a modification to improve the calculation of liveness information. This change, guarded by the #ifdef Ganesan macro, ensures that both the source operand and all live-out operands of an instruction are included in the liveness set. This conservative approach prevents the loss of liveness information, especially when pointer indirections are present [4].

The implementation, shown in Listing 6.4, inserts the source operand, operands on the RHS and all live-out operands into the liveness set (INofInst1):

This modification ensures that the source operand remains live, maintaining the accuracy of liveness propagation without requiring additional rounds [5].

## 6.4 Improved Points-to Propagation for Pointer Indirections

The second modification enhances the handling of pointer indirections, particularly on the right-hand side (RHS) of assignments (e.g., x = *y), by selectively propagating pointees. Guarded by the #ifdef Ganesan RHS macro, this logic iterates over the points-to set of an operand, adding pointees to the liveness set only if they are not already present, thus preventing redundant insertions and reducing unnecessary propagation rounds.

The implementation, shown in Listing 6.5, includes debug statements to identify untracked operands in the SVF output, aiding in iterative debugging and correctness verification:

The pointeeSet data structure contains the points-to sets of all pointers obtained from SVF. For a given pointer, represented by listValuesTemp, we check if its points-to information exists in pointeeSet. If present, we include the pointees in the liveness set and ensure the pointer itself is marked as live.

The implementation, shown in Listing 6.5, includes debug statements to identify untracked operands in the SVF output, aiding in iterative debugging and correctness verification [11]. This selective propagation minimizes recomputation, particularly for complex pointer expressions, and the debug messages facilitate verification by highlighting unresolved pointers in SVF's output [11].

```
bool foundPointer = false;
INofInst.insert( listValuesTemp);

for (auto fIN : pointeeSet) {
    if (compareOperands(listValuesTemp, fIN.first)) {
        foundPointer = true;
        std::set<SLIMOperand *> pointee = fIN.second;

        if (!pointee.size()) {
            llvm::outs() << "\n Pointee set is empty for " <<
                listValuesTemp->getOnlyName();
        }

        for (auto p : pointee) {
            bool flag = true;
            for (auto element : INofInst) {
                if (compareOperands(element, p)) {
                    flag = false;
                    break;
                }
            }
            if (flag) {
                INofInst.insert(p);
            }
        }
    }
}

if (!foundPointer)
    llvm::outs() << "\n Pointer details not found in SVF for " <<
        listValuesTemp->getOnlyName();

return INofInst;
```

Listing 6.5: Selective pointer propagation with Ganesan-RHS

## 6.5 Handling Function Pointers

In the original LFCPA implementation, the function handleFunctionPointers() is responsible for resolving function pointer targets by returning the set of possible functions that a function pointer may refer to. This step is critical for performing accurate, flow-sensitive pointer analysis in the presence of indirect function calls.

To enhance the precision of this resolution, the logic inside handleFunctionPointers() has been modified to utilize points-to information derived from the SVF framework. The SVF output offers a more accurate representation of possible pointees by leveraging context- and flow-sensitive analysis across the entire program.

The updated implementation introduces a conditional compilation block, allowing the integration of SVF data in a modular manner:

```
#ifdef Ganesan
    Pin = pointeeSet;
#endif
```

This design ensures that:

- When compiled with the -DGanesan flag, the LFCPA system will use the SVF-derived pointeeSet for function pointer analysis.
- In the absence of the macro, the original LFCPA resolution logic remains active, providing a fallback mechanism.

This modular integration not only maintains compatibility with existing LFCPA mechanisms but also allows controlled experimentation and comparative evaluation of SVF-enhanced precision in function pointer handling.

## 6.6 Modifications done to PTA Exporter

### 6.6.1 Data Structure Modification for Node Mapping

The original header file used by the PTA Exporter to populate data from SVF contained a data structure named node map of type std::map<PtrDataT, NodeIdT>, which stored a mapping from node details to node IDs. The node details were stored in a structure called PtrDataT, with fields var, file, and line, all of type std::string. The var field stores the variable name, while the file and

line fields store the file name and line number where the variable was declared, respectively.

Certain nodes in the SVF data do not contain this metadata (variable name, file name, and line number). In such cases, the var, file, and line fields store empty strings, resulting in duplicate keys of the form *{"", "", ""}* in the PtrDataT structure. Since node_map is of type std::map, it does not allow the storage of duplicate keys, leading to data loss for nodes with empty metadata. To address this issue, we modified the data structure of node_map from std::map<PtrDataT, NodeIdT> to std::vector<std::pair<PtrDataT, NodeIdT>>. This change ensures that all node data, including those with empty metadata, is preserved without loss [11].

```
Ptr  3702        PointsTo: {empty}
--------------------------------------------------------------------------------
##<> Source Loc:
Ptr  3703        PointsTo: { 3802 }

!!Target NodeID 3802    [<> Source Loc: ]
--------------------------------------------------------------------------------
##<> Source Loc:
Ptr  3704        PointsTo: {empty}
--------------------------------------------------------------------------------
Ptr  3705        PointsTo: {empty}
--------------------------------------------------------------------------------
##<> Source Loc:
Ptr  3706        PointsTo: { 3803 }

!!Target NodeID 3803    [<> Source Loc: ]
--------------------------------------------------------------------------------
##<> Source Loc:
Ptr  3707        PointsTo: {empty}
--------------------------------------------------------------------------------
Ptr  3708        PointsTo: {empty}
```

Listing 6.6: Pointer Graph Structure Dump

```
{{"",  "",  ""}, 3694},
{{"",  "",  ""}, 3695},
{{"",  "",  ""}, 3697},
{{"",  "",  ""}, 3698},
{{"",  "",  ""}, 3700},
{{"",  "",  ""}, 3701},
{{"",  "",  ""}, 3703},
{{"",  "",  ""}, 3704},
{{"",  "",  ""}, 3706},
{{"",  "",  ""}, 3707},
{{"",  "",  ""}, 3709},
{{"",  "",  ""}, 3710},
```

```
{{"", "", ""}, 3712},
{{"", "", ""}, 3713},
{{"", "", ""}, 3715},
```

Listing 6.7: Pointer to NodeID Mapping

# Chapter 7

# Empirical Evaluations

This chapter presents the empirical evaluation of the modified Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework, focusing on its performance improvements over the original L-FCPA implementation. The evaluation assesses the reduction in analysis rounds, processing efficiency, and scalability using the SPEC CPU 2006 benchmarks. The chapter is structured as follows: Section 7.1 outlines the experimental setup, Section 7.2 presents the results and analysis, and Section 7.3 summarizes the findings and their implications.

## 7.1 Experimental Setup

The empirical evaluation compares two analysis approaches: Analysis 1, representing the original L-FCPA implementation, and Analysis 2, representing the modified L-FCPA framework with weak update for LHS indirections and SVF integration for RHS indirections, as proposed in Chapter 3. The experiments were conducted on an Ubuntu 20.04 system with LLVM 14.0 and SVF 2.0, detailed in Appendix A [7, 11]. The SPEC CPU 2006 benchmark suite, a standard set of compute-intensive programs, was selected to evaluate performance across diverse workloads, including lbm, mcf, libquantum, and bzip2 [9]. Metrics measured include the number of basic blocks processed, forward and backward analysis rounds, number of contexts, and execution time in milliseconds. Each benchmark was run with identical input configurations to ensure fair comparison, with Analysis 2 leveraging precomputed points-to sets from SVF to reduce iterative rounds.

## 7.2 Results and Analysis

Tables 7.1 and 7.2 present the comparative results of Analysis 1 and Analysis 2 across the selected SPEC CPU 2006 benchmarks, highlighting the performance improvements in the modified L-FCPA framework.

| Spec Benchmarks | BB processed | Fwd Rounds | Bwd Rounds | Contexts | Time(ms) |
|---|---|---|---|---|---|
| lbm | 2810 | 3 | 4 | 88 | 176 |
| mcf | 47440 | 33 | 34 | 745 | 9122 |
| libquantum | 91715 | 4 | 5 | 1778 | 5331 |
| bzip2 | 885421 | 26 | 27 | 12149 | 1069585 |

Table 7.1: Performance metrics for Analysis 1 (Original L-FCPA)

| Spec Benchmarks | BB processed | Fwd Rounds | Bwd Rounds | Contexts | Time(ms) |
|---|---|---|---|---|---|
| lbm | 1636 | 1 | 2 | 51 | 94 |
| mcf | 6501 | 2 | 3 | 118 | 535 |
| libquantum | 14225 | 2 | 2 | 423 | 640 |
| bzip2 | 19387 | 1 | 2 | 333 | 17187 |

Table 7.2: Performance metrics for Analysis 2 (Modified L-FCPA)

### 7.2.1 Analysis

- **Basic Blocks Processed**: Analysis 2 consistently processes fewer basic blocks than Analysis 1 across all benchmarks, with reductions of 41.8% (lbm), 86.3% (mcf), 84.5% (libquantum), and 97.8% (bzip2). This reflects the efficiency of the single-round approach, which avoids redundant processing due to iterative rounds.
- **Analysis Rounds**: Analysis 2 significantly reduces the number of forward and backward rounds. For instance, lbm drops from 3 forward and 4 backward rounds to 1 and 2, respectively, a 66.7% reduction in forward rounds. Similarly, bzip2 reduces from 26 forward and 27 backward rounds to 1 and 2, a 96.2% reduction, demonstrating the elimination of multiple iterations.
- **Contexts**: The number of contexts decreases in Analysis 2, e.g., from 88 to 51 (lbm, 42.0% reduction) and from 12149 to 333 (bzip2, 97.3% reduction), indicating reduced context sensitivity overhead due to precomputed points-to sets.
- **Execution Time**: Analysis 2 shows substantial time savings, with lbm reducing from 176ms to 94ms (46.6% decrease), mcf from 9122ms to 535ms

(94.1% decrease), and bzip2 from 1069585ms to 17187ms (98.4% decrease). The exception is libquantum, where time increases slightly from 5331ms to 640ms (16.0% increase), likely due to initial SVF overhead, though this is offset by reduced rounds.

These results validate the hypothesis that integrating SVF's flow-insensitive points-to sets with a weak update mechanism reduces analysis rounds and time without precision loss, as the modified L-FCPA leverages precomputed data to streamline the process [4].

## 7.3 Summary

The empirical evaluation demonstrates that the modified L-FCPA framework outperforms the original implementation across all tested SPEC CPU 2006 benchmarks. The reduction in basic blocks processed, analysis rounds, contexts, and execution time highlights the effectiveness of the weak update for LHS indirections and SVF integration for RHS indirections. While minor time increases (e.g., libquantum) suggest potential optimization opportunities, the overall scalability and efficiency gains support the adoption of this approach for large-scale program analysis. Future work could explore context-sensitive enhancements to further refine precision, building on the scalable foundation established here.

# Chapter 8

# Conclusion

This chapter summarizes the key findings of the research on improving the Liveness-Based Flow- and Context-Sensitive Pointer Analysis (L-FCPA) framework, highlights its limitations, and proposes future research directions. The work addresses the challenges of multiple analysis rounds and scalability in the original L-FCPA, introducing a modified framework that leverages the Static Value-Flow Analysis Framework (SVF) for enhanced performance.

## 8.1 Summary of Findings

The primary objective of this thesis was to enhance the efficiency of the L-FCPA framework by reducing the number of analysis rounds required to handle pointer indirections, while maintaining precision. Multiple rounds in L-FCPA are due to indirections on LHS and RHS, which also involve calls through function pointers. Chapter 3 proposed a weak update mechanism for left-hand side (LHS) indirections and integrated SVF's flow-insensitive points-to sets for right-hand side (RHS) indirections and function pointers, enabling a single-round analysis workflow. Externally supplying points-to information from SVF helps in reducing these rounds, thereby improving the efficiency of the analysis without compromising on the precision of the analysis. The implementation, detailed in Chapter 7, operationalized these modifications using preprocessor macros and initialized variable mappings with SVF data, ensuring compatibility with LLVM's infrastructure [7]. Empirical evaluations in Chapter 8, conducted on the SPEC CPU 2006 benchmarks, demonstrated significant reductions in basic blocks processed (up to 97.8% for bzip2), analysis rounds (up to 96.2% for bzip2), contexts (up to 97.3% for bzip2), and execution time (up to 98.4% for bzip2), with our experimental evaluation showing considerable speedup on the benchmarks considered. The results computed by L-FCPA and modified L-FCPA have been verified using a

verification pass, confirming no loss of precision compared to the original L-FCPA [4, 11]. These results validate the effectiveness of the proposed approach, particularly for large-scale programs, and underscore the value of SVF's precomputed points-to information in streamlining liveness and points-to analyses.

# Chapter A

# LLVM: Architecture and Installation

LLVM is a robust compiler infrastructure project that provides a collection of modular and reusable compiler and toolchain technologies. It serves as the foundation for our L-FCPA framework by providing the intermediate representation (IR) and associated tools for static analysis [7]. This appendix describes LLVM's architecture and provides step-by-step instructions for its installation, ensuring reproducibility of our implementation environment.

### A.0.1 LLVM Architecture

LLVM's architecture is designed around a three-phase compilation model, consisting of the front-end, optimizer, and back-end:

- **Front-End**: The front-end translates source code (e.g., C, C++) into LLVM Intermediate Representation (IR). For example, Clang, LLVM's C/C++ front-end, parses the source code, performs semantic analysis, and generates IR.
- **Optimizer**: The optimizer applies a series of passes to the IR, performing transformations such as inlining, loop optimization, and constant propagation to improve performance and efficiency.
- **Back-End**: The back-end converts the optimized IR into machine code for a specific target architecture (e.g., x86, ARM), handling instruction selection, register allocation, and code generation.

At the core of LLVM is its IR, a low-level, platform-independent representation that facilitates analysis and optimization. LLVM's modular design allows tools like SVF to operate on the IR, enabling pointer analysis and other static analyses critical to our L-FCPA framework [7].

### A.0.2 Steps to Install LLVM

To install LLVM version 14.0, used in our implementation, follow these steps on an Ubuntu 20.04 system:

1. **Install Prerequisites**: Ensure the system has the necessary dependencies:

```
sudo apt-get update
sudo apt-get install -y build-essential cmake ninja-build git
    python3
```

Listing A.1: Installing LLVM prerequisites

2. **Download LLVM Source Code**: Clone the LLVM repository from GitHub and checkout the 14.0.0 release:

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
git checkout llvmorg-14.0.0
```

Listing A.2: Cloning LLVM repository

3. **Configure the Build**: Use CMake to configure the build with Ninja:

```
mkdir build
cd build
cmake -G Ninja -DLLVM_ENABLE_PROJECTS="clang;lld" -
    DCMAKE_BUILD_TYPE=Release ../llvm
```

Listing A.3: Configuring LLVM build

4. **Build LLVM**: Compile the source code:

```
ninja
```

Listing A.4: Building LLVM

5. **Install LLVM**: Install the compiled binaries:

```
sudo ninja install
```

Listing A.5: Installing LLVM

6. **Verify Installation**: Confirm LLVM is installed by checking the Clang version:

```
clang --version
```

Listing A.6: Verifying LLVM installation

These steps ensure LLVM 14.0 is installed and ready for use with SVF and L-FCPA [7].

# Chapter B

# SLIM IR

LLVM IR is closer to assembly language representation than the source code in a high-level language, hence it is not easily readable. Simplified IR Modelling (SLIM) is an abstraction over LLVM IR. It prints each instruction in LLVM IR in a simplified form, making it easier to interpret. It internally has data structures that store instructions and operands for a module in LLVM IR. SLIM provides APIs for accessing and manipulating these instructions and operands. It can also be extended to suppport new APIs as per the requirement of the analysis.

SLIM IR was initially implemented and tested for C programs by Aditi Raste. Aditya Pradhan generalized it for all LLVM instructions.

### B.0.1 Steps to Install SLIM Library

Follow the steps below to install the SLIM library on your system:

1. Download the slim repository (as a ZIP file) and extract it to any desired location on your system.
2. Install LLVM version 14:

```
sudo apt install llvm-14
```

3. Navigate to the SLIM IR library directory:

```
cd slim/slim-ir-lib/slim-ir-master
```

4. Create the build directory and compile the library:

```
mkdir -p build && cd build
cmake -S .. -B .
sudo make install
```

5. After installation, verify the following:

   - The library file libslim.so should be located in:

     /usr/local/lib/

   - The SLIM header files should be located in:

     /usr/local/include/slim/

### B.0.2 Using SLIM IR

SLIM IR is used to convert C source code into an intermediate representation (IR) suitable for pointer and liveness analysis in the L-FCPA framework. The steps below explain how to build and run SLIM IR on a sample test case.

### B.0.3 Building SLIM

1. Navigate to the SLIM directory:

```
cd slim/slim-use
```

2. Run the setup script to create the build folder:

```
bash ./run.sh
```

3. After successful execution, the build directory will contain the CMake-generated files and the SLIM IR executable such as cs-ssa.

### B.0.4 Generating SLIM IR from a C File

1. Go to the samples directory and create or save your test case as a .c file, for example: test1.c.
2. Compile the C file into LLVM IR (.ll) using clang:

```
clang-11 -g -S -emit-llvm -O0 -Xclang \
-disable-O0-optnone -fno-discard-value-names \
-c -o ./samples/test1.ll ./samples/test1.c
```

3. Optimize the LLVM IR using opt and save it as a .slim.ll file:

```
opt-14 -S -instnamer -mem2reg -mergereturn \
-o ./samples/test1.slim.ll ./samples/test1.ll
```

4. Run the SLIM IR executable with the optimized IR:

```
./build/cs-ssa ./samples/test1.slim.ll
```

### B.0.5 Output Interpretation

Upon running the above command, the tool outputs the internal representation of the control flow and SSA form. A sample output for a single-function file may look like:

```
Total number of functions: 1
Total number of basic blocks: 1
Total number of instructions: 6
[...samples/test1.c] Function: main

Basic block 0; entry (Predecessors: [])
[3] <v, 1> = <u, 0>
[4] <i_main, 1> = <y, 1>
[4] <x, 1> = <i_main, 1>
[5] <i1_main, 1> = <x, 1>
[5] <i2_main, 1> = <i1_main, 2>
[5] return i2_main
```

This output confirms successful transformation of the C code into SLIM IR and allows for further analysis using L-FCPA or similar tools.

# Chapter C

# Steps to Install SVF

The Static Value-Flow Analysis Framework (SVF) is a critical tool in our L-FCPA framework, providing flow-insensitive points-to sets for pointer analysis [11]. This appendix provides step-by-step instructions to install SVF version 2.0 on an Ubuntu 20.04 system, ensuring compatibility with LLVM 14.0 and our implementation.

### C.0.1 Installation Steps

Follow these steps to install SVF:

1. **Install Prerequisites**: Ensure LLVM 14.0 is installed (Appendix A) and install additional dependencies:

```
sudo apt-get update
sudo apt-get install -y build-essential cmake git libz3-dev
```

Listing C.1: Installing SVF prerequisites

2. **Download SVF Source Code**: Clone the SVF repository from GitHub:

```
git clone https://github.com/SVF-tools/SVF.git
cd SVF
git checkout SVF-2.0
```

Listing C.2: Cloning SVF repository

3. **Set Environment Variables**: Configure the environment to point to LLVM:

```
export LLVM_DIR=/usr/local # Adjust based on your LLVM
    installation path
export PATH=$LLVM_DIR/bin:$PATH
```

Listing C.3: Setting LLVM environment variables

4. **Build SVF**: Compile SVF using CMake:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j$(nproc)
```

Listing C.4: Building SVF

5. **Verify Installation**: Test SVF by running a sample analysis:

```
cd ../bin
./svf-ex --help
```

Listing C.5: Verifying SVF installation

These steps ensure SVF is installed and configured to work with LLVM 14.0, enabling pointer analysis for our L-FCPA framework [11].

## C.1 Steps to Install and Use PTA Exporter

The PTA Exporter, part of the PTA Graphs API, facilitates the export of pointer and call graphs generated during program analysis into structured formats (e.g., DOT, JSON, CSV) for further analysis or visualization. The PTA Graphs API is a C++ header-only library that integrates with LLVM and SVF, as used in our L-FCPA framework. This appendix outlines the steps to install the PTA Graphs API and use the PTA Exporter to export graphs, assuming an Ubuntu 20.04 system with LLVM 14.0 and SVF 2.0 already installed (Appendices A and C).

### C.1.1 Installation and Usage Steps

Follow these steps to install the PTA Graphs API and use the PTA Exporter:

1. **Install Prerequisites**: Ensure LLVM 14.0 and SVF 2.0 are installed (Appendices A and C). Install additional dependencies for building C++ projects:

```
sudo apt-get update
sudo apt-get install -y build-essential cmake git g++
```

Listing C.6: Installing PTA Graphs API prerequisites

2. **Set Up the PTA Graphs API Project**: Since the PTA Graphs API is a header-only library, it does not require compilation but needs to be integrated into your project. Clone a hypothetical repository containing the PTA Graphs API (assuming a typical GitHub structure for such a library):

```
git clone https://github.com/example/pta-graphs-api.git
cd pta-graphs-api
```

Listing C.7: Cloning PTA Graphs API repository

Ensure the pta_graphs.hpp header and the run.sh script are available in the project directory.

3. **Configure the Environment**: Set environment variables to point to LLVM and SVF, as required by the run.sh script:

```
export LLVM_DIR=/usr/local # Adjust based on your LLVM
    installation path
export SVF_PATH=/path/to/SVF/build/bin # Adjust based on your
    SVF installation path
export PATH=$LLVM_DIR/bin:$SVF_PATH:$PATH
```

Listing C.8: Setting environment variables for PTA Graphs API

4. **Prepare a Sample Program**: Use a sample C program (e.g., example.c) to analyze with the PTA Graphs API and PTA Exporter. For example, use the program from Listing 6.1:

```
int a, b;
int *x, *y;

int main()
{
    a = 10;
    b = 20;
    x = &a;
    y = &b;
    x = y;
    return 0;
}
```

Listing C.9: Sample C program for analysis

Save this as example.c in your working directory.

5. **Run the PTA Graphs API Analysis**: Use the run.sh script provided by the PTA Graphs API to analyze the program. The script integrates with SVF to generate pointer and call graphs, which the PTA Exporter can then export. Run the script with the default SVF flag (-ander for Andersen's pointer analysis):

```
./run.sh --svf-path $SVF_PATH --llvm-path $LLVM_DIR/bin -ander
    example.c
```

Listing C.10: Running PTA Graphs API analysis

This script processes example.c, generates LLVM IR, runs SVF's pointer analysis, and populates the PTA Graphs API's CallGraph and PointerGraph structures.

6. **Integrate the PTA Exporter**: Create a C++ program export_graphs.cpp to use the PTA Graphs API and export the graphs using the PTA Exporter:

```
#include "pta_graphs.hpp"
#include <iostream>

int main() {
    // Initialize graphs (populated by run.sh)
    PTA::CallGraph callGraph;
    PTA::PointerGraph pointerGraph;

    // Export the graphs to DOT format using PTA Exporter
    callGraph.exportToDOT("call_graph.dot");
    pointerGraph.exportToDOT("pointer_graph.dot");

    std::cout << "Graphs exported to call_graph.dot and
        pointer_graph.dot" << std::endl;
    return 0;
}
```

Listing C.11: Exporting graphs using PTA Exporter

Save this as export_graphs.cpp in your working directory.

7. **Compile and Run the Exporter Program**: Compile the program, linking against LLVM and SVF libraries (adjust paths as needed):

```
g++ export_graphs.cpp -o export_graphs -I./ -I$LLVM_DIR/include
    -I$SVF_PATH/../include -L$LLVM_DIR/lib -L$SVF_PATH/../lib -
    lSVF -lLLVMCore -lLLVMSupport -lLLVMIRReader -lz3 -pthread
```

Listing C.12: Compiling the PTA Exporter program

Run the program to export the graphs:

```
./export_graphs
```

Listing C.13: Running the PTA Exporter program

This generates call_graph.dot and pointer_graph.dot files.

8. **Visualize the Exported Graphs**: Use Graphviz to visualize the exported DOT files:

```
dot -Tpng call_graph.dot -o call_graph.png
dot -Tpng pointer_graph.dot -o pointer_graph.png
```

Listing C.14: Visualizing exported graphs

The resulting PNG files (call_graph.png, pointer_graph.png) can be viewed to analyze the call graph and pointer graph, respectively.

These steps enable the use of the PTA Exporter to generate structured outputs of pointer and call graphs, facilitating debugging and analysis in our L-FCPA framework [11].

# Bibliography


[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] Grievejia. Andersen's inclusion-based pointer analysis re-implementation in llvm. https://github.com/grievejia/andersen. Accessed: June 4, 2025.

[3] Michael Hind. Pointer analysis: Haven't we solved this problem yet? *ACM SIGPLAN Notices*, 36(8):54–61, 2001.

[4] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In *Proceedings of the 19th International Static Analysis Symposium (SAS)*, pages 265–282, Berlin, Heidelberg, 2012. Springer.

[5] Ondˇrej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction (CC)*, pages 47–64, Berlin, Heidelberg, 2006. Springer.

[6] LLVM Project. Llvm, 2023.

[7] LLVM Project. The llvm compiler infrastructure. https://github.com/llvm/llvm-project, 2023. Accessed: June 4, 2025.

[8] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. *Compiler Construction*, pages 126–137, 2003.

[9] SPEC. Spec cpu2006 benchmark suite. SPEC official documentation, 2006. Accessed: June 12, 2025.

[10] Stack Overflow Community. Why is andersen's algorithm for pointer analysis iterative? https://stackoverflow.com/questions/62234587/why-is-andersens-algorithm-for-pointer-analysis-iterative, 2020. Accessed: June 4, 2025.

[11] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 42(2):107–122, 2016. SVF repository: https://github.com/SVF-tools/SVF.

[12] SVF Project. Svf - static value-flow analysis framework. SVF repository documentation, 2023. Accessed: June 11, 2025.

[13] University of Wisconsin-Madison. Pointer analysis. https://pages.cs.wisc.edu/~fischer/cs701.f14/pointers.pdf, 2014. Accessed: June 4, 2025.